Using Streaming SIMD Extensions 2 (SSE2) in a Double-precision 3D Transform

Version 2.0

7/00

Order Number 248604-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Pentium® 4 processor update	7/00
1.0	Original publication of document	9/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

Alan Watt, *3D Computer Graphics 2nd Edition*, Addison Wesley, 1993.

James Foley, Andries van Dam, Steven Feiner, John Hughes, *Computer Graphics: Principles and Practice, 2nd Edition*, Addison Wesley, 1990.

Intel Corporation, *C++ Class Libraries for SIMD Operation Reference Manual*, order number 693500, 1999.

Intel Corporation, *Intel® C\C++ Compiler User's Guide*, order number 741901, 1999.

Using Streaming SIMD Extensions 2 (SSE2) for SAXPY/DAXPY, Intel Application Note, AP-935, Copyright 2000

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) technology introduces new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced with the Streaming SIMD Extensions (SSE). The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application. This application note describes the implementation of a double-precision 3D geometry transformation, and includes code examples that exploit the SSE2 instructions.

Optimization techniques of several implementations using the 128-bit XMM registers to work with packed double-precision floating-point data and related SSE2 instructions are compared to provide developers the tools for optimizing their own implementations of the double-precision 3D transformation. Source code containing various implementations of the algorithm is included.

2 Double-precision 3D Transformation

While the 3D transformation is an integral component of every 3D-geometry engine, the double-precision implementation of the algorithm is typically reserved for use in applications where numerically accurate modeling is paramount. The objective of the transform is to convert an object's vertex values from its *local coordinate space*—defined about a coordinate system containing only the given object—into *world coordinate space*—a coordinate system defined to include all objects in a given 3D scene. The transformation itself is a simple multiplication of a 4-by-4 transformation matrix by a 4-by-1 vertex vector in its local space, resulting in a new 4-by-1 vertex in world space (see Figure 1) [Watt, 45-49].

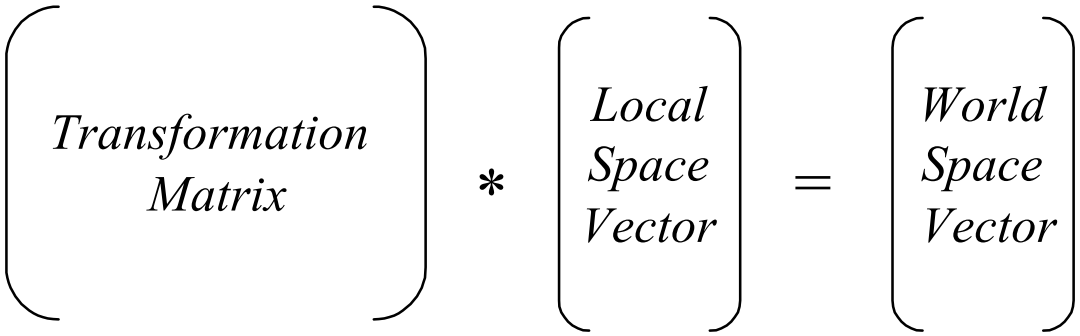

Figure 1: The 3D Transform is the multiplication of a 4-by-4 matrix by a 4-by-1 vector

2.1 Applications for Double-precision 3D Transformation

3D graphics can be applied to an unlimited variety of applications ranging from 3D games to interplanetary gravitational research. Different applications have different requirements in their 3D transformation implementations. 3D games, for example, focus principally on the demand for real-time rendering. For reasons of performance, games most often employ a system that requires only single-precision floating-point values to adequately "convince" the human eye that the image portrayed on the screen is accurate.

Double-precision implementations of the transform are found primarily in modeling systems and applications where accurate modeling of object rotation, translation, and scaling is imperative. Double-precision accuracy is most useful in high-end 3D modeling, animation, scientific research applications, computer aided design (CAD), and medical visualization. While the optimization techniques discussed in this application note are widely applicable to most double-precision graphics systems, slight modifications may necessary to accommodate the specific needs of any given system.

2.2 Background of Double-precision 3D Transformation

The field of 3D graphics has long used polygonal modeling to create three-dimensional objects that interact in 3D environments known as *scenes*. In polygonal modeling, graphical objects are defined by a list of vertices that form the shape of the object. The role of the geometry engine is to perform mathematical operations on the vertex lists, enabling a realistic modeling of the interactions between objects, or between objects and light sources, scene boundaries, and even the laws of physics.

Mathematicians such as René Descartes and Sir Isaac Newton formulated the founding principles of 3D graphics as early as the 17th century. But it wasn't until the 1960's and 1970's that vector and raster displays were invented, bringing to life the implementation of polygonal modeling via the microcomputer. Probably the most complete and comprehensive work on the subject of computer graphics to date is *Computer Graphics: Principles and Practice, 2nd Edition* by James Foley, et al [Foley].

2.3 Implementing the Double-precision 3D Transformation

Two different approaches to vertex data structuring were taken in conjunction with this application note. The first approach is the standard array-of-structures (AoS) format in which a structure is defined to contain the four double-precision values *x, y, z,* and *w*, and an array declared to contain some quantity, *n*, of such structures (see Figure 2). This is probably the simplest approach because developers are most accustomed to the format, and it yields decent performance gains with relatively little effort. If the aim of the application is to reduce development time, and not necessarily to gain every ounce of performance, this approach may be most fitting.

```
struct AOS
{
      double x, y, z, w;
};

AOS Vertex[n];
```

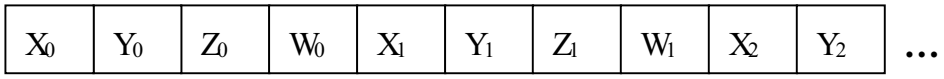

Figure 2: The layout and organization of the array-of-structures data format

The second, less common data structure is a structure of arrays (SoA) format in which four arrays of vertex data are laid out in "planar" format. The first array contains some quantity, *n*, of X values, the second contains *n* Y values, the third, *n* Z values, and the fourth, *n* W values (see Figure 3). This structure will usually require some initial data re-formatting, but is effective in reducing the number of unnecessary loads and cache evictions, and is, therefore, more optimal than the array-of-structures (AoS) format. The structure-of-arrays (SoA) data organization, as it is known, allows the developer to take full advantage of the parallelism inherent in the SSE2 instructions, by operating on two double-precision data elements at a time. See the included source code for data structure examples.

```
struct SOA
{
      double x[n], y[n], z[n], w[n];
};

SOA Vertex;
```

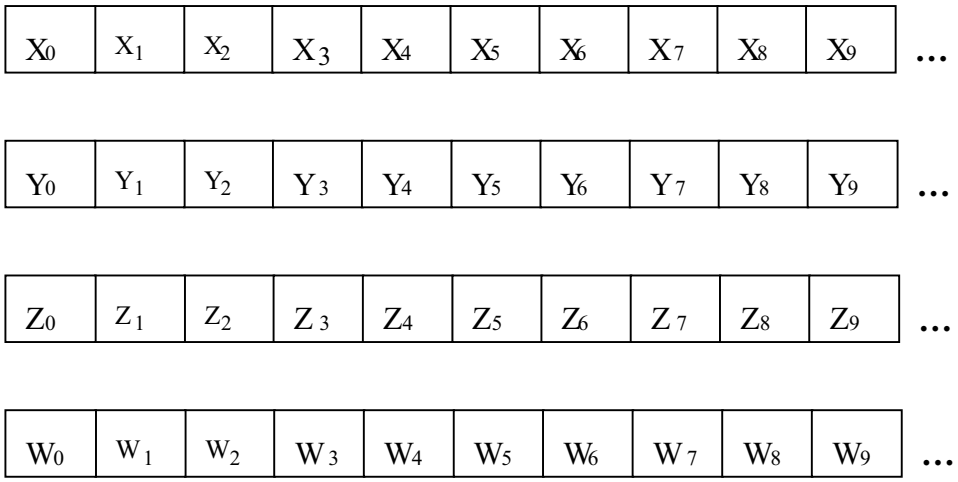

Figure 3: The layout and organization of the structures-of-arrays data format

2.3.1 Techniques

Many 3D graphics engines use vertex coordinates X, Y, Z, and W, where W is always assumed to be 1.0 in local coordinate space. An optimization common to these engines is to perform a modified matrix multiplication in which X, Y, and Z are multiplied by the corresponding matrix elements, while the fourth-column matrix element is merely added in, rather than wasting a multiplication step with a multiplicative-identity value of 1.0. Examples of this technique can be seen in all of the code samples included here and should be taken into consideration when optimizing a geometry pipeline.

The optimized versions of the 3D transform use the 128-bit registers to perform two transformations per loop. In cases where the structure-of-arrays (SoA) data structure is used, two consecutive vertex values can be loaded simultaneously using the `movapd` instruction (i.e. X_0 and X_1 are consecutive in memory and can be accessed with one memory reference). When the array-of-structures (AoS) format is used,

vertex values must be loaded separately using the `movhpd` and `movlpd` instructions. X_0 and X_1 are still loaded, but because they are no longer contiguous in memory, they must be accessed individually. The SoA approach is obviously the fastest, but the AoS approach also produces good performance gains, and should not be overlooked.

2.3.2 Tips and Tricks

The transform uses each X, Y, and Z input value to compute the four new vertex values X', Y', Z', and W'. Matrix elements, on the other hand, are loaded, used once per loop, and can then be discarded. Since register pressure is not an issue with the transform, it is possible to improve performance by keeping vertex values in the registers for the duration of an entire loop. By forcing the registers containing matrix values to be the destination of each operation, vertex data is loaded only once at the

```
movapd    xmm0,  [ebx+eax]          ; load x+1|x
movapd    xmm1,  [esi+0]            ; load m00|m00
mulpd     xmm1,  xmm0

movapd    xmm2,  [ecx+eax]          ; load y+1|y
movapd    xmm3,  [esi+16]           ; load m01|m01
mulpd     xmm3,  xmm2

movapd    xmm4,  [edx+eax]          ; load z+1|z
movapd    xmm5,  [esi+32]           ; load m02|m02
mulpd     xmm5,  xmm4
addpd     xmm3,  xmm1

movapd    xmm7,  [esi+48]           ; load m03|m03
addpd     xmm5,  xmm3
addpd     xmm7,  xmm5
movapd    [ebx+eax],     xmm7       ; store (x+1)'|x'

/*  x, y, and z values are still available in their original registers */
movapd    xmm1,  [esi+64]           ; load m10|m10
mulpd     xmm1,  xmm0

movapd    xmm3,  [esi+80]           ; load m11|m11
mulpd     xmm3,  xmm2

          • • •
```

Figure 4: Optimized assembly listing showing the re-use of X, Y, and Z values

beginning of the loop and overwritten at the beginning of the next loop. This approach can be seen in the abbreviated assembly code listing shown in Figure 4.

Accessing double-precision data with the intrinsics versions of the instructions requires using a memory pointer to the double-precision value being accessed. While it is tempting to rewrite a loop using existing structures in conjunction with the address-of operator (&), doing so may force the compiler to assume that memory pointers are aliased to the same data structure. As a result, the compiler may disable optimizations that could have been performed if the address-of operator had not been used (see Figure 5).

```
AOS vertex[n];
        for (i = 0; i < n; i++)
        {
              __m128d tx, m00;
              tx = _mm_load_pd(tx, &(vertex[i].x));
              m00 = _mm_load_pd(m00, &(Matrix->m00));
              m00 = _mm_mul_pd(m00, tx);
        };
```

Figure 5: Misuse of address-of operator (&) disables some intrinsics optimizations

To enable all optimizations, loops optimized with intrinsics should be redesigned to be addressable without the address-of operator. The simplest approach to eliminating the address-of operator is to cast the pointer to the array of `double` data elements to a pointer to an array of type `__m128d` (see Figure 6). This approach works best when the data organization is SoA, but can also be used with the AoS formatting with some overhead for shuffling data around.

```
        AOS vertex[n];
        __m128d *V = vertex;
        __m128d *M = Matrix;
        for (i = 0; i < n; i++)
        {
              __m128d m00;
              m00 = _mm_mul_pd(M[0], V[i]);
        };
```

Figure 6 – Casting to a __m128d array enables intrinsics optimizations

3 Performance

Transformation results vary depending on data structure and coding technique. This section briefly discusses the trade-off between development time and performance gain for the 3D transformation. Three separate methods for implementing double-precision SSE2 instructions are explored. The most basic is inline assembly coding, which places the responsibility of register allocation and instruction scheduling on the shoulders of the programmer. The other three approaches—using the intrinsics for the Intel® C\C++ Compiler, using the C++ Class Libraries for SIMD Operations, and using the vectorizer of the Intel C\C++ Compiler —place allocation and scheduling responsibility on the compiler, allowing the programmer to concentrate on higher-level algorithmic modifications. Regardless of the chosen approach, the developer will most likely see significant performance improvements in the implementation of the double-precision 3D transform with SSE2 instructions if the suggestions in this application note are observed.

3.1 Gains/Improvements

Starting with a standard C implementation and AoS data formatting as the baseline, the transform was rewritten in C code using the SoA format. Both code segments are included in Section 5 C/C++ Coding Examples on page 13. Intrinsics were then coded for both data formats, using the techniques and tips discussed in Section 2.3.2. Each of the intrinsics versions showed significant improvement over the standard C implementations, and as expected, the SoA intrinsics implementation performed somewhat better than its AoS counterpart. See Section 7 SSE2 Intrinsics Coding Examples on page 16 for a complete code listing of the intrinsics implementations of the 3D transform.

For those who prefer C++, an implementation of the SoA code was written using the C++ Class Libraries for SIMD Operations. The class libraries are as easy to use as any other C++ class, but are really an abstraction of the intrinsics. By choosing to implement the vector classes, developers stand to gain many coding benefits. The code is as easy to write as C++ code, performs on par with the intrinsics version of the same code, and looks as clean as the original C implementation of the code. The C++ vector class implementation is shown in Section 6.

Another implementation uses the vectorization capabilities of the Intel C\C++ compiler to compile the C/C++ code to use the SIMD instructions. The SOA 3D Transform implementation is a good candidate for vectorization. To learn more about vectorization, the reader is referred to app note: AP-935, Using Streaming SIMD Extensions 2 (SSE2) for SAXPY/DAXPY. The author of the SAXPY/DAXPY app note gives a brief discussion on how to use the vectorizer. The reader is also referred to the Intel Compiler User Guide for a more general discussion on using the vectorizer. The vectorizer code is shown in section 8.

Finally, assembly versions were hand-coded to offer a performance comparison. In these small test cases, the assembly did marginally outperform the intrinsics code. Developers should realize, however, that by implementing inline assembly in a full-scale application, they may be removing opportunities for the compiler to perform inter-procedural optimizations while increasing the cost of interleaving and inlining other code segments with the hand-coded functions. Often by seeking the last few percent gain that assembly optimization affords the developer, the application loses a few percent of compiler optimizations, resulting in a time-consuming wash. The developer should weigh the additional time required to code inline assembly against its performance gain and make an informed decision. If the gain is not greater than a few percent, it is probably better simply to implement intrinsics. Those interested in hand-optimizing assembly instructions should refer to section 8 for complete code listings of the assembly language implementations.

3.2 Considerations

While the results in this paper show the SoA data format be the optimal structure, there is another data structure not explored here that may produce higher performance under certain conditions. Suppose that the array size is very large for the SoA input data. If the array size is so large that X, Y, and Z values are each contained in a separate page of memory, a page miss will occur every time a vertex component is accessed. To ensure that vertex components exist on the same page, data can be both packed *and* interleaved in what is known as the *hybrid* structure-of-arrays format (see Figure 7). The hybrid data order provides the best of both worlds by allowing like data elements to be accessed simultaneously from memory, while avoiding the page miss problems that occur with large arrays of planar data. If the developer experiences lower than expected memory performance with other approaches, investigating the hybrid structure may be an alternative.

```
struct Hybrid_SOA
{
      double x[2], y[2], z[2], w[2];
};

Hybrid_SOA Vertex[n/2];
```

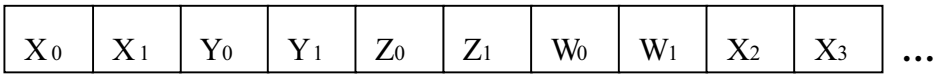

Figure 7: "Hybrid" structure-of-arrays data format may be a more optimal alternative

4 Conclusion

Whether implemented with array-of-structures (AoS) or structure-of-arrays (SoA) data formatting, the SSE2 instructions offer significant performance gains for the double-precision 3D transform. This application note shows that use of the SSE2 instructions with the SoA data structure offers notable speedups over SSE2 instructions with the AoS format, even though both are significantly faster than a standard C implementation. Even if the move to planar data structures is not feasible for an application, significant speedups can be gained through use of the SSE2 instructions and should be pursued.

Although some developers may require the performance gained by programming with assembly instructions, most will achieve the highest return-on-investment by optimizing their standard C implementations using the SSE2 intrinsics, by optimizing their standard C++ implementations using the C++ Class Libraries for SIMD Operations, or by using vectorization. You should modify the techniques shown in this application note to suit your specific implementation of the 3D transformation. The results will be well worth the small effort spent.

5 C/C++ Coding Examples

```
struct Matrix
{
     DBL m00, m01, m02, m03;
     DBL m10, m11, m12, m13;
     DBL m20, m21, m22, m23;
     DBL m30, m31, m32, m33;
} M44;

struct Soa
{
     DBL *x, *y, *z, *w;

     void Initialize()
     {
       x = _mm_malloc(NUM_ELEMENTS*sizeof(DBL), 32);
       y = _mm_malloc(NUM_ELEMENTS*sizeof(DBL), 32);
       z = _mm_malloc(NUM_ELEMENTS*sizeof(DBL), 32);
       w = _mm_malloc(NUM_ELEMENTS*sizeof(DBL), 32);
     }
} SOA_Vertex;

__declspec(align(32))
struct Aos
{
     DBL x, y, z, w;
} AOS_Vertex[NUM_ELEMENTS];

int AOS_Transform ()
{
  Aos *vertex = AOS_Vertex;

  for (int i=0; i < length; i++)
  {
     DBL tx, ty, tz, tw;
```

```
     tx = M44->m00*vertex[i].x + M44->m01*vertex[i].y + M44->m02*vertex[i].z
+ M44->m03;
     ty = M44->m10*vertex[i].x + M44->m11*vertex[i].y + M44->m12*vertex[i].z
+ M44->m13;
     tz = M44->m20*vertex[i].x + M44->m21*vertex[i].y + M44->m22*vertex[i].z
+ M44->m23;
     tw = M44->m30*vertex[i].x + M44->m31*vertex[i].y + M44->m32*vertex[i].z
+ M44->m33;

     vertex[i].x = tx; vertex[i].y = ty; vertex[i].z = tz; vertex[i].w = tw;
   }
}

int SOA_Transform()
{
   for (int i=0; i < length; i++)
   {
     DBL tx, ty, tz, tw;

     tx = M44->m00*SOA_Vertex.x[i] + M44->m01*SOA_Vertex.y[i] +
            M44->m02*SOA_Vertex.z[i] + M44->m03;
     ty = M44->m10*SOA_Vertex.x[i] + M44->m11*SOA_Vertex.y[i] +
            M44->m12*SOA_Vertex.z[i] + M44->m13;
     tz = M44->m20*SOA_Vertex.x[i] + M44->m21*SOA_Vertex.y[i] +
            M44->m22*SOA_Vertex.z[i] + M44->m23;
     tw = M44->m30*SOA_Vertex.x[i] + M44->m31*SOA_Vertex.y[i] +
            M44->m32*SOA_Vertex.z[i] + M44->m33;

     SOA_Vertex.x[i] = tx; SOA_Vertex.y[i] = ty; SOA_Vertex.z[i] = tz;
SOA_Vertex.w[i] = tw;
   }
}
```

6 SSE2 C++ Vector Class Coding Example

```
struct WMatrix
{
      union
      {
            struct
            {
               DBL m00, m00p, m01, m01p, m02, m02p, m03, m03p;
               DBL m10, m10p, m11, m11p, m12, m12p, m13, m13p;
               DBL m20, m20p, m21, m21p, m22, m22p, m23, m23p;
               DBL m30, m30p, m31, m31p, m32, m32p, m33, m33p;
            };
            struct
            {
               __m128d dm00, dm01, dm02, dm03;
               __m128d dm10, dm11, dm12, dm13;
               __m128d dm20, dm21, dm22, dm23;
               __m128d dm30, dm31, dm32, dm33;
            };
      };
} WM44;


int VectorClass_SOA()
{
      F64vec2 *x = SOA_Vertex.x; F64vec2 *y = SOA_Vertex.y; F64vec2 *z =
SOA_Vertex.z;
      F64vec2 *w = SOA_Vertex.w; F64vec2 *WM = WM44;


      for (int i=0; i < length/2; i++)
      {
            F64vec2 tx, ty, tz, tw;
            tx = x[i] * WM[0] + y[i] * WM[1] + z[i] * WM[2] + WM[3];
            ty = x[i] * WM[4] + y[i] * WM[5] + z[i] * WM[6] + WM[7];
            tz = x[i] * WM[8] + y[i] * WM[9] + z[i] * WM[10] + WM[11];
            tw = x[i] * WM[12] + y[i] * WM[13] + z[i] * WM[14] + WM[15];
            x[i] = tx; y[i] = ty; z[i] = tz; w[i] = tw;
      }
      return EXIT_SUCCESS;
}
```

7 SSE2 Intrinsics Coding Examples

```
int AOS_Intrinsics()
{
     DBL *vertex = AOS_Vertex;

     for (int i=0; i < length; i+=2)
     {
          __m128dtx, ty, tz, tw;
          __m128dmx0, mx1, mx2, mx3;

          // Compute (x) and (x+1)
          tx = _mm_loadl_pd(tx, vertex+i*4+0);
          tx = _mm_loadh_pd(tx, vertex+i*4+4);
          mx0 = _mm_mul_pd(tx, WM->dm00);

          ty = _mm_loadl_pd(ty, vertex+i*4+1);
          ty = _mm_loadh_pd(ty, vertex+i*4+5);
          mx1 = _mm_mul_pd(ty, WM->dm01);

          tz = _mm_loadl_pd(tz, vertex+i*4+2);
          tz = _mm_loadh_pd(tz, vertex+i*4+6);
          mx2 = _mm_mul_pd(tz, WM->dm02);
          mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm03)));
          _mm_storel_pd(vertex+i*4+0, mx0);
          _mm_storeh_pd(vertex+i*4+4, mx0);

          // Compute (y) and (y+1)
          mx0 = _mm_mul_pd(tx, WM->dm10);
          mx1 = _mm_mul_pd(ty, WM->dm11);
          mx2 = _mm_mul_pd(tz, WM->dm12);
          mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm13)));
          _mm_storel_pd(vertex+i*4+1, mx0);
          _mm_storeh_pd(vertex+i*4+5, mx0);

          // Compute (z) and (z+1)
          mx0 = _mm_mul_pd(tx, WM->dm20);
          mx1 = _mm_mul_pd(ty, WM->dm21);
```

```
            mx2 = _mm_mul_pd(tz, WM->dm22);
            mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm23)));
            _mm_storel_pd(vertex+i*4+2, mx0);
            _mm_storeh_pd(vertex+i*4+6, mx0);

            // Compute (w) and (w+1)
            mx0 = _mm_mul_pd(tx, WM->dm30);
            mx1 = _mm_mul_pd(ty, WM->dm31);
            mx2 = _mm_mul_pd(tz, WM->dm32);
            mx0= _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm33)));
            _mm_storel_pd(vertex+i*4+3, mx0);
            _mm_storeh_pd(vertex+i*4+7, mx0);
        }
}


int SOA_Intrinsics()
{
        Soa vertex = SOA_Vertex;

        for (int i=0; i < length; i+=2)
        {
            __m128dtx, ty, tz, tw;
            __m128dmx0, mx1, mx2, mx3;

            // Compute (x) and (x+1)
            tx = _mm_load_pd(vertex.x+i);
            mx0 = _mm_mul_pd(tx, WM->dm00);
            ty = _mm_load_pd(vertex.y+i);
            mx1 = _mm_mul_pd(ty, WM->dm01);
            tz = _mm_load_pd(vertex.z+i);
            mx2 = _mm_mul_pd(tz, WM->dm02);
            mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm03)));
            _mm_store_pd(vertex.x+i, mx0);

            // Compute (y) and (y+1)
            mx0 = _mm_mul_pd(tx, WM->dm10);
            mx1 = _mm_mul_pd(ty, WM->dm11);
```

```
            mx2 = _mm_mul_pd(tz, WM->dm12);
            mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm13)));
            _mm_store_pd(vertex.y+i, mx0);

            // Compute (z) and (z+1)
            mx0 = _mm_mul_pd(tx, WM->dm20);
            mx1 = _mm_mul_pd(ty, WM->dm21);
            mx2 = _mm_mul_pd(tz, WM->dm22);
            mx0 = _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm23)));
            _mm_store_pd(vertex.z+i, mx0);

            // Compute (w) and (w+1)
            mx0 = _mm_mul_pd(tx, WM->dm30);
            mx1 = _mm_mul_pd(ty, WM->dm31);
            mx2 = _mm_mul_pd(tz, WM->dm32);
            mx0= _mm_add_pd(mx0, _mm_add_pd(mx1, _mm_add_pd(mx2, WM->dm33)));
            _mm_store_pd(vertex.w+i, mx0);
        }
    }
```

8 SSE2 Assembly Coding Examples

```
int AOS_Asm()
{
     int  len = length;
     Aos *vertex = AOS_Vertex;
     WMatrix  *WM = WM44;

     __asm
     {
       xor       eax, eax
       mov     edi, len
       imul    edi, 32
       sub       edi, 8
       mov     ecx, vertex
       mov     edx, WM

LOOP:
       cmp     eax, edi
       jge       END_LOOP

       // Compute (x) and (x+1)
       movlpd    xmm0, [ecx+eax+0]       ; load x+1|x
       movhpd xmm0, [ecx+eax+32]
       movapd    xmm1,  [edx+0]        ; load m00|m00
       mulpd     xmm1,  xmm0

       movlpd    xmm2,  [ecx+eax+8]      ; load y+1|y
       movhpd xmm2,  [ecx+eax+40]
       movapd    xmm3,  [edx+16]       ; load m01|m01
       mulpd     xmm3,  xmm2
       movlpd    xmm4, [ecx+eax+16]      ; load z+1|z
       movhpd xmm4, [ecx+eax+48]
       movapd    xmm5,  [edx+32]       ; load m02|m02
       mulpd     xmm5,  xmm4
       addpd     xmm3,  xmm1
       movapd    xmm7,  [edx+48]       ; load m03|m03
       addpd     xmm5,  xmm3
```

```
        addpd     xmm7,   xmm5
        movlpd    [ecx+eax+0],   xmm7      ; store X'
        movhpd [ecx+eax+32],  xmm7      ; store (X+1)'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
        // Compute (y) and (y+1)
        movapd    xmm1,   [edx+64]      ; load m10|m10
        mulpd     xmm1,   xmm0
        movapd    xmm3,   [edx+80]      ; load m11|m11
        mulpd     xmm3,   xmm2
        movapd    xmm5,   [edx+96]      ; load m12|m12
        mulpd     xmm5,   xmm4
        addpd     xmm3,   xmm1
        movapd    xmm7,   [edx+112]     ; load m13|m13
        addpd     xmm5,   xmm3
        addpd     xmm7,   xmm5
        movlpd    [ecx+eax+8],   xmm7      ; store Y'
        movhpd [ecx+eax+40],  xmm7      ; store (Y+1)'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
        // Compute (z) and (z+1)
        movapd    xmm1,   [edx+128]     ; load m20|m20
        mulpd     xmm1,   xmm0
        movapd    xmm3,   [edx+144]     ; load m21|m21
        mulpd     xmm3,   xmm2
        movapd    xmm5,   [edx+160]     ; load m22|m22
        mulpd     xmm5,   xmm4
        addpd     xmm3,   xmm1
        movapd    xmm7,   [edx+176]     ; load m23|m23
        addpd     xmm5,   xmm3
        addpd     xmm7,   xmm5
        movlpd    [ecx+eax+16],  xmm7      ; store Z'
        movhpd [ecx+eax+48],  xmm7      ; store (Z+1)'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
        // Compute (w) and (w+1)
```

```
        movapd    xmm1,  [edx+192]     ; load m30|m30
        mulpd     xmm1,  xmm0
        movapd    xmm3,  [edx+208]     ; load m31|m31
        mulpd     xmm3,  xmm2
        movapd    xmm5,  [edx+224]     ; load m32|m32
        mulpd     xmm5,  xmm4
        addpd     xmm3,  xmm1
        movapd    xmm7,  [edx+240]     ; load m33|m33
        addpd     xmm5,  xmm3
        addpd     xmm7,  xmm5
        movlpd [ecx+eax+24], xmm7      ; store W'
        movhpd [ecx+eax+56], xmm7      ; store (W+1)'

        add       eax, 64
        jmp       LOOP
END_LOOP:
     }
}

int SOA_Asm()
{
     int  len = length*8;
     Soa vertex = SOA_Vertex;
     WMatrix  *WM = WM44;

     __asm
     {
        xor    eax, eax
        mov    ebx, vertex.x
        mov    ecx, vertex.y
        mov    edx, vertex.z
        mov    edi, vertex.w
        mov    esi, WM

LOOP:
        cmp    eax, len
        jge    END_LOOP
```

```
        // Compute (x) and (x+1)
        movapd    xmm0,  [ebx+eax]     ; load x+1|x
        movapd    xmm1,  [esi+0]       ; load m00|m00
        mulpd     xmm1,  xmm0
        movapd    xmm2,  [ecx+eax]     ; load y+1|y
        movapd    xmm3,  [esi+16]      ; load m01|m01
        mulpd     xmm3,  xmm2
        movapd    xmm4,  [edx+eax]     ; load z+1|z
        movapd    xmm5,  [esi+32]      ; load m02|m02
        mulpd     xmm5,  xmm4

        addpd     xmm3,  xmm1
        movapd    xmm7,  [esi+48]      ; load m03|m03
        addpd     xmm5,  xmm3
        addpd     xmm7,  xmm5
        movapd    [ebx+eax],  xmm7     ; store (x+1)'|x'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
        // Compute (y) and (y+1)
        movapd    xmm1,  [esi+64]      ; load m10|m10
        mulpd     xmm1,  xmm0
        movapd    xmm3,  [esi+80]      ; load m11|m11
        mulpd     xmm3,  xmm2
        movapd    xmm5,  [esi+96]      ; load m12|m12
        mulpd     xmm5,  xmm4
        addpd     xmm3,  xmm1
        movapd    xmm7,  [esi+112]     ; load m13|m13
        addpd     xmm5,  xmm3
        addpd     xmm7,  xmm5
        movapd    [ecx+eax],  xmm7     ; store (y+1)'|y'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
        // Compute (z) and (z+1)
        movapd    xmm1,  [esi+128]     ; load m20|m20
        mulpd     xmm1,  xmm0
        movapd    xmm3,  [esi+144]     ; load m21|m21
```

```
        mulpd     xmm3,   xmm2
        movapd    xmm5,   [esi+160]     ; load m22|m22
        mulpd     xmm5,   xmm4
        addpd     xmm3,   xmm1
        movapd    xmm7,   [esi+176]     ; load m23|m23
        addpd     xmm5,   xmm3
        addpd     xmm7,   xmm5
        movapd    [edx+eax],  xmm7      ; store (z+1)'|z'

/* x, y, and z values are still available in their original registers to
eliminate increased load port usage */
        // Compute (w) and (w+1)
        movapd    xmm1,   [esi+192]     ; load m30|m30
        mulpd     xmm1,   xmm0
        movapd    xmm3,   [esi+208]     ; load m31|m31
        mulpd     xmm3,   xmm2
        movapd    xmm5,   [esi+224]     ; load m32|m32
        mulpd     xmm5,   xmm4
        addpd     xmm3,   xmm1
        movapd    xmm7,   [esi+240]     ; load m33|m33
        addpd     xmm5,   xmm3
        addpd     xmm7,   xmm5
        movapd    [edi+eax],  xmm7      ; store (w+1)'|w'

        add       eax, 16
        jmp       LOOP
END_LOOP:
     }
}
```

9 Vectorizing Compiler Code Example

```
int VecAOS ()
{
     Aos *vertex = AOS_Vertex;
     int loc_length = length;
     double m00 = M44->m00, m01 = M44->m01, m02 = M44->m02, m03 = M44->m03,
           m10 = M44->m10, m11 = M44->m11, m12 = M44->m12, m13 = M44->m13,
           m20 = M44->m20, m21 = M44->m21, m22 = M44->m22, m23 = M44->m23,
           m30 = M44->m30, m31 = M44->m31, m32 = M44->m32, m33 = M44->m33;

     #pragma ivdep
     #pragma vector aligned
     for (int i=0; i < loc_length; i++)
     {
          DBL tx, ty, tz, tw;

          tx = m00*vertex[i].x + m01*vertex[i].y + m02*vertex[i].z + m03;
          ty = m10*vertex[i].x + m11*vertex[i].y + m12*vertex[i].z + m13;
          tz = m20*vertex[i].x + m21*vertex[i].y + m22*vertex[i].z + m23;
          tw = m30*vertex[i].x + m31*vertex[i].y + m32*vertex[i].z + m33;

          vertex[i].x = tx; vertex[i].y = ty; vertex[i].z = tz;
          vertex[i].w = tw;
     }

     return EXIT_SUCCESS;
}
```

```
int VecSOA ()
{
     int loc_length = length;
     double m00 = M44->m00, m01 = M44->m01, m02 = M44->m02, m03 = M44->m03,
          m10 = M44->m10, m11 = M44->m11, m12 = M44->m12, m13 = M44->m13,
          m20 = M44->m20, m21 = M44->m21, m22 = M44->m22, m23 = M44->m23,
          m30 = M44->m30, m31 = M44->m31, m32 = M44->m32, m33 = M44->m33;
     double *x = SOA_Vertex.x, *y = SOA_Vertex.y, *z = SOA_Vertex.z,
     *w = SOA_Vertex.w;

     #pragma ivdep
     #pragma vector aligned
     for (int i=0; i < loc_length; i++)
     {
          DBL tx, ty, tz, tw;

          tx = m00*x[i] + m01*y[i] + m02*z[i] + m03;
          ty = m10*x[i] + m11*y[i] + m12*z[i] + m13;
          tz = m20*x[i] + m21*y[i] + m22*z[i] + m23;
          tw = m30*x[i] + m31*y[i] + m32*z[i] + m33;

          x[i] = tx;
          y[i] = ty;
          z[i] = tz;
          w[i] = tw;
     }

     return EXIT_SUCCESS;
}
```

Appendix A - Performance Data

Performance Data Revision History

Revision	Revision History	Date
2.0	Updated with 1.2 GHz Pentium® 4 processor performance data	7/00
1.0	Original publication of document	9/99

Table 1: Performance Data of 3D Transform Implementations

Performance Data in Nano-seconds		
Cases	**Pentium® III Processor (733 MHz)**	**Pentium 4 Processor (1.2 GHz)**
C Code AOS	60.1	29.9
C Code SOA	63.3	32.4
ASM SSE2 AOS		18.5
ASM SSE2 SOA		14.6
Vector Classes SSE2 SOA		15.4
Intrinsics SSE2 AOS		19.4
Intrinsics SSE2 SOA		16.1
Vectorizer SSE2 AOS		19.4
Vectorizer SSE2 SOA		13.7

Table 2: Speedups from Table 1 Performance Data

Implementations and Platforms	Speedup
Pentium 4 Processor (Vectorizer SSE2 SOA vs. C Code AOS)	2.18
C Code AOS on Pentium 4 Processor vs. C Code AOS on Pentium III Processor	2.01

Performance was measured using a Pentium III 733 MHz processor and a Pentium 4 1.2 GHz processor. See Test Systems Configuration on page A-3 for a detailed description of the test systems.

Performance was first measured with all the data in the level one cache (perfect cache). The results of the tests are summarized in Table 1. Vector lengths were chosen to be equal to 1,000 as an arbitrary average size-per-object for graphics standards at the time of this writing. For each element of the vector, 6 floating-point operations (three multiplies and three adds) were performed.

For the double-precision 3D transform, one would expect SSE2 instructions to provide a 2x improvement over scalar code. This is due to the 2-wide nature of the registers, accomplishing twice as much work with the same number of instructions as the original C code. This is proven to be true in the array-of-structures (AOS) format where the SSE2 implementations are 1.5 – 1.6x faster than the baseline C code.

As would be expected, the structure-of-arrays (SOA) implementations were even faster, showing that speeds as high as 2.18x are obtainable over the baseline code. This is due to the more streamlined accesses that are available thanks to the modified memory structure. By accessing two data elements per memory read, the overhead of shuffling is eliminated and the pipelined micro-architecture able to continue with fewer stalls due to memory latencies. Notice the SOA vectorizer implementation is faster than the ASM implementation. In this case, the vectorizer code not only provided the fastest way to use the SIMD instructions but it also provided the best performance.

Test Systems Configuration

Table 3: Pentium III Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 4: Pentium 4 Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195